Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 99-004

Access Region Locality for High-Bandwidth Processor Memory System Design

Sangyeun Cho, Pen-chung Yew, and Gyungho Lee

February 15, 1999

Access Region Locality for High-Bandwidth Processor Memory System Design

Sangyeun Cho, Pen-Chung Yew, and Gyungho Lee[†]

Dept. of Comp. Sci. and Eng.
University of Minnesota
Minneapolis, MN 55455

[†]Division of Engineering
University of Texas at San Antonio
San Antonio, TX 78249

E-mail: sycho@cs.umn.edu

Abstract

This paper explores an important behavior of memory access instructions, called access region locality. Unlike the traditional temporal and spatial data locality that focuses on individual memory locations and how accesses to the locations are inter-related, the access region locality concerns with each static memory instruction and its range of access locations at run time. We consider program's data, heap, and stack regions in this paper. Our experimental study using a set of SPEC95 benchmark programs shows that most memory reference instructions access a single region at run time. Also shown is that it is possible to predict the access region of a memory instruction accurately at run time by scrutinizing the addressing mode of the instruction and the past access region history of it. An important implication of the access region locality is that two memory accesses to different regions are data independent. Utilizing this property, we evaluate a novel processor memory system and pipeline design which can provide high data cache bandwidth without increasing the critical path of the processor, by decoupling the memory instructions that access the program's stack at an early pipeline stage. Experimental results indicate that the proposed technique achieves comparable or better performance than a conventional memory design with a heavily multi-ported data cache that can lead to much higher hardware complexity.

Keywords: data locality, instruction-level parallelism, run-time stack, data stream partitioning, processor pipeline, superscalar processor

1 Introduction

Technological and architectural innovations have enabled development of powerful microprocessors that can execute several instructions concurrently at a very high clock rate [9, 30, 10]. These processors select and execute independent instructions at run time, assisted by hardware mechanisms for control speculation, register renaming, and data-flow execution [12]. With ample on-chip hardware resources that will become available within a few years, researchers are actively proposing even more aggressive microarchitectures that can issue up to 16 or more instructions in a cycle [13, 19, 22]. To increase the exploitable *instruction level parallelism* (ILP) by better utilizing the available hardware parallelism, various techniques to speculate on control [15, 31], data values [14], and data dependences [16] are being pursued.

In a future wide-issue processor with aggressive control and data speculation techniques, efficient handling of memory references will become a more critical factor that affects the overall performance. Cache memories [23] have been used in virtually all recent microprocessors to shorten the average memory access latency. Temporal and spatial localities are two important operating principles for various cache memories. In addition to the memory latency problem caused by the processor-memory speed gap, the ability to provide enough memory bandwidth (or cache ports) to the processor core is extremely important for a future wide-issue processor to achieve its full performance potential [25, 21, 3]. For example, for a processor to sustain ten instructions per cycle (IPC), the memory subsystem should provide a minimum bandwidth of four references per cycle, or more, to prevent excessive queuing delays, assuming that about 40% of all instructions are loads and stores [13].

This paper explores a very useful behavior of memory reference instructions for a high-bandwidth processor memory system design, called *access region locality*. The access region locality states that a memory reference instruction typically accesses a single region[1] at run time and (thus) the region it accesses is highly predictable. An important implication of the access region locality is that any two memory references that access distinct regions are data independent. Figure 1 presents a small code section to illustrate the access region of memory reference instructions.

Four (static) memory references of interest are highlighted: `b[i]` in line 8, `c[i]` in line 9, `*parm1` in line 10, and `a` in line 13. When the function `foo ()` is called, `b[i]` accesses the heap region and `c[i]`

[1] An access region **R** is defined as (**L**,**U**), where **L** is the lower bound on the address of the accessed locations of a memory reference instruction at run time and **U** is the upper bound. Program's data, heap, and stack segments are regions.

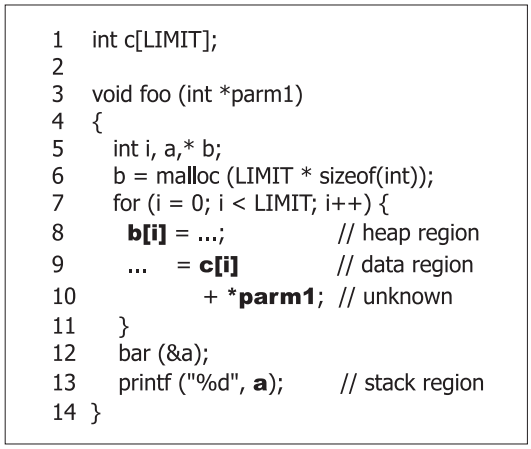

Figure 1: An example code segment that shows memory references to different regions.

accesses the data region, determined by where b[] and c[] are allocated (see line 6 and 1). The access region of *parm1 is unknown from the given code segment. It can access any region at run time, depending on the address passed from the call site of the foo () function. Since a is a local variable whose address is taken (in line 12), the reference to a becomes a stack access.

We show in Section 3 that memory reference instructions which can access more than one region, such as *parm1, are few, and their access regions are accurately predictable at run time using a simple predictor similar to the ones used for branch prediction. Even without program's high-level information, *i.e.*, given only the binary code, a simple one-bit predictor can classify the memory references into stack and non-stack references with an accuracy of over 99.6% on average.

Based on the access region locality, we study a novel processor pipeline and memory system design called *data-decoupled architecture* [3]. The data-decoupled architecture divides the data memory stream into two data-independent streams in an early pipeline stage using a prediction mechanism and feeds each stream to a separate memory unit. The separation of independent memory references facilitates the use of dual caches with a smaller number of ports, each of which is associated with a dedicated pool of reservation stations.

The data-decoupled approach to the memory system design can have two crucial advantages over a conventional design when used in a wide-issue superscalar processor. First, the cost and the complexity of building a large cache with many ports is reduced. Implementing a reasonably-sized data cache with more than two ports becomes increasingly inefficient. That is, such a cache can occupy significantly more chip area, and/or can have longer access latency [21, 3]. More importantly, the

network and the control logic for orchestrating memory accesses between a large number of reservation stations and cache ports become simpler. Such reduction in hardware complexity can lead to a shorter clock cycle time [18]. Second, partitioning memory references can facilitate more specialized handling of each partitioned stream. Fast forwarding described in Section 4 is one such technique. Our performance results based on an aggressive superscalar processor show that the data-decoupled architecture achieves a similar or better performance than a conventional memory system design with a heavily multi-ported data cache.

The rest of this paper is organized as follows. Section 2 summarizes previous related work. Section 3 studies the access region locality using some experimental data and develops a prediction mechanism. Section 4 discusses the data-decoupled architecture and gives preliminary evaluation results. The conclusions are summarized in Section 5.

2 Related Work

Designing an effective multi-ported data cache has been a topic of active research as aggressive multiple-issue processors emerge. Sohi and Franklin [25] first predicted that the L1 cache bandwidth will eventually become a performance bottleneck for a wide-issue processor and proposed a non-blocking, multi-ported data cache design with interleaved banks as a solution. Wilson *et al.* [29] argued that adding more ports to the L1 cache can become costly and/or inefficient in terms of space and time. As an alternative to a dual-ported cache design found in some recent microprocessors, they proposed augmenting a small *line buffer* to a single-ported data cache to effectively increase the port efficiency. Rivers *et al.* [21] also studied the impact of using a line buffer per bank in a wide, interleaved cache. These studies have focused on increasing the efficiency of cache ports by adding a small buffer, or understanding tradeoffs of various strategies in terms of the cost and performance under specific processor models. The data-decoupled architecture is largely orthogonal to the data cache design techniques. Our approach focuses on relieving the hardware complexity involved in the large instruction window, network, and data cache by splitting the memory instructions early. It allows the use of dual data caches attached to a separate instruction window each, putting less burden on bandwidth requirements of the caches.

The idea of separately servicing stack accesses is not new. Ditzel and McLellan [4] studied a

transparent data buffer as a close mapping of the run-time stack, called the *stack cache*. The stack cache is effectively a large register file to simulate the run-time stack that replaces the general register file. The *contour buffer* proposed by Flynn and Hoevel [8] in their *Directly Executed Languages* model, is a programmer-addressable buffer that is used in conjunction with the run-time stack in memory. Stanley and Wedig [26] proposed three buffer management algorithms for a Top of Stack (TOS) buffer, which is a register file designed to cache the top elements of the stack. These studies aimed primarily to reduce the impact of a procedure call/return on the processor performance, motivated by an observation that programs written in a high-level language tend to have many procedure calls and returns [6], and that a function call is the most costly source language statement [20]. Our work mainly focuses on relieving the data cache bandwidth by offloading the stack accesses, rather than trying to reduce the number of memory references or the access latency. Unlike the previous approaches, the technique proposed in this paper does not require processor intervention or complex algorithms to manage the buffer, which were mandated in the previous techniques to deal with buffer overflow/underflow and context switches. The proposed LVC (Local Variable Cache) [3] under the data-decoupled architecture framework fits in the memory hierarchy, and the data movement between the LVC and lower-level memory is initiated automatically.

There are dynamic techniques to decouple a portion of data references and service them using separate, specialized functional units. Lipasti introduced a notion called *load stream partitioning* in his Superflow processor model [13], which partitions loads into multiple streams based on their run-time behavior, and sends them to disjoint functional units for processing. The functional units used include a constant verification unit, a queue for load/store folding, a stream buffer/prefetch engine, and a conventional data cache. Techniques to detect dependent memory access instructions and explicitly synchronize and forward data between them have been proposed [28, 17]. They provide a dynamic technique to detect a producer operation and a consumer operation within the instruction window, and try to forward the data in a special buffer before the effective addresses are calculated, without accessing the cache memory. We haven't studied the performance of the data-decoupled architecture and the above proposals in an integrated framework for comparison. In terms of hardware complexity, however, the above proposals – their partitioning algorithms and the network that connects different functional units and reservation stations – may become quite high as the processor's issue bandwidth increases.

3 Access Region Locality

A program's memory space is divided into a few regions or segments: data, heap, and stack regions under a typical run-time system [1].[2] We study in this section how each memory reference instruction accesses memory regions using a profiling tool and a set of benchmark programs. We first study how each static memory instruction accesses regions at run time. Then we design and evaluate a run-time prediction mechanism. This section will serve as a basis for the discussions in Section 4.

3.1 Methodology

We use a memory reference profiler based on the Simplescalar tool set [2] for the results reported in this section. In each simulated cycle, it fetches and executes one instruction as specified in the program. While doing so, it collects desired information, *i.e.*, which region(s) a memory reference instruction accesses.

We use eight integer and four floating-point programs from the SPEC95 benchmark suite [27], whose characteristics are summarized in Table 1. *101.tomcatv*, *102.swim*, *103.su2cor*, and *107.mgrid* are floating-point programs. All the programs were compiled using EGCS[3] version 1.1b [5] at the -O3 optimization level with loop unrolling. Either *train* or *test* input is used in most cases, with some data set modification to control the simulation time.

3.2 Per-reference memory access behavior

3.2.1 Access regions and access region locality

We analyze what region(s) each memory instruction accesses in a program execution. Depending on the accessed region(s), instructions are classified into 7 different classes, as shown in Figure 2. It is observed that majority of memory instructions (in "D", "H", and "S" classes) reference a single region at run time. Only an average of 1.8% and 1.9% of all the static instructions access more than one region in the integer and floating-point programs studied, respectively. Although varied from one program to another, these instructions account for 0% – 9.6% of all the dynamic memory references.

[2]Program's text region is yet another memory region. Accesses to the text region are directed to a separate instruction cache in many recent processors.

[3]EGCS is based on widely used GCC. It has a global CSE pass and a global instruction scheduling pass additionally, assisted by an improved alias analysis algorithm.

Benchmark	Input	Inst. count	Loads	Stores
099.go	train	541M	22%	8%
124.m88ksim	ref	250M	14%	8%
126.gcc	stmt-protoize.i	220M	22%	14%
129.compress	train (100K)	293M	21%	13%
130.li	ctak.lsp	434M	28%	19%
132.ijpeg	penguin.ppm	621M	19%	9%
134.perl	scrabbl.pl	525M	26%	15%
147.vortex	train (1 iter.)	284M	29%	22%
101.tomcatv	test ($N = 253$, 1 iter.)	549M	21%	12%
102.swim	test (3 iter.)	473M	22%	8%
103.su2cor	test	676M	23%	10%
107.mgrid	train (1 iter.)	684M	32%	6%

Table 1: Input, dynamic instruction count, and percentage of load and store instructions in each benchmark program. Percentage of load or store instructions is relative to the total instruction count.

Programs such as *124.m88ksim, 134.perl* and *101.tomcatv* have more instructions that access multiple regions than other programs.

The strong correspondence between memory instructions and their accessed memory regions is a natural consequence of how programs are written. Most memory instructions access either a fixed location (*e.g.*, static variables and temporary local variables) or a set of locations that belong to (instances of) a single data structure, such as an array or C structure, that is allocated in a pre-determined region. Therefore, even when it is difficult to predict the exact address of a memory reference, it is still feasible to predict its access region, as will be shown in this section.

Integer programs except *099.go* have a significant number of heap-accessing instructions as they allocate many data structures dynamically, while floating-programs do not. It is interesting to note that although each class varies much in its size, the sum of the "D" and "H" classes remains roughly comparable across programs. Over 50% of all the static memory instructions only access program's stack region at run time on average. These instructions are for passing procedure parameters, spilling and reloading registers, and storing local variables. The static and dynamic distribution of the instructions from different classes will be determined by the writing style of the programmer, the programming language used, the underlying processor architecture, and how the compiler generates memory reference instructions, e.g., during register allocation, etc.

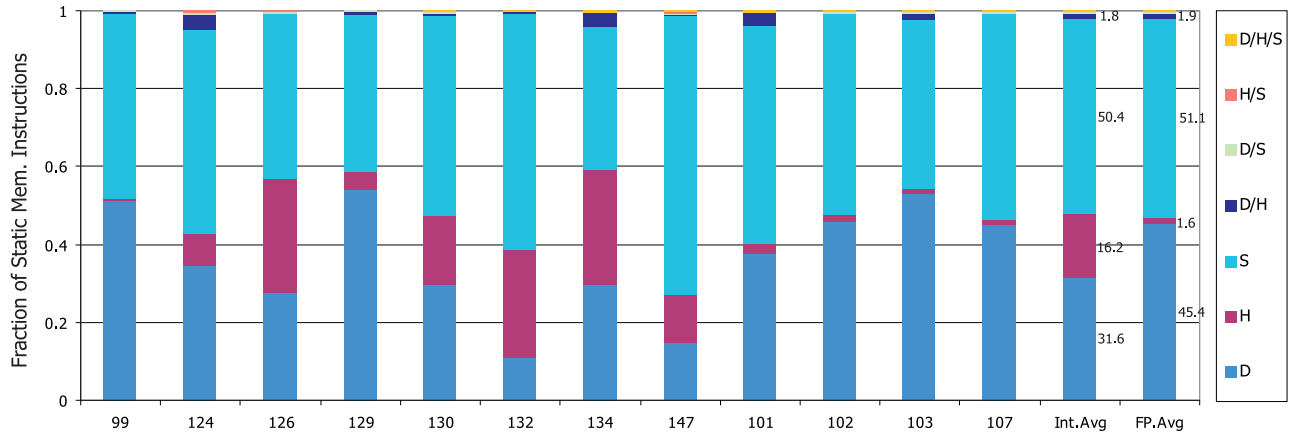

Figure 2: Breakdown of static memory instructions based on the region(s) they access at run time. "D" stands for data region, "H" heap region, and "S" stack region. "D/S" denotes the instructions that access both the data and the stack regions when the program is executed.

3.2.2 Interleaving of accesses to different regions

The next important questions for the data-decoupled architecture to be effective are how many dynamic references are directed to each access region and how accesses to different regions are interleaved. The information provides a notable insight into how much memory bandwidth (or how many cache ports) is required by the memory accesses toward each region. It is also useful in estimating the performance impact when separate cache ports are provided for a certain memory region.

To answer the questions, we counted the number of memory references in the last 32 or 64 instructions (32 or 64-wide sliding instruction window) each cycle. Table 2 reports the average and standard deviation of the counted numbers. The standard deviation shows how bursty a group of memory references is; the higher it is, the more bursty the memory references are. The standard deviation becomes large when the occurrences of memory accesses are clustered.

Three observations are made from the result. First, either data or stack accesses consume more memory bandwidth than the other two types of accesses in all the programs studied. There are six programs (*099.go*, *124.m88ksim*, *129.compress*, *102.swim*, *103.su2cor*, and *107.mgrid*) that have more data accesses than heap or stack accesses. The other six programs have more stack references than data or heap references. All the programs except one (*124.m88ksim*) have fewer heap references than stack references.

Second, comparing the average number of accesses and the standard deviation, accesses to the

	Window Size = 32			Window Size = 64		
Benchmark	Data	Heap	Stack	Data	Heap	Stack
099.go	6.11 (2.71)	0.00 (0.00)	3.61 (4.62)	12.23 (4.37)	0.00 (0.00)	7.23 (7.83)
124.m88ksim	2.91 (2.45)	2.14 (3.69)	1.90 (2.20)	5.82 (2.18)	4.29 (7.21)	3.81 (3.35)
126.gcc	3.48 (4.23)	1.69 (2.36)	6.45 (5.13)	6.96 (7.97)	3.38 (4.09)	12.91 (8.54)
129.compress	9.94 (3.70)	0.00 (0.02)	1.08 (1.50)	19.86 (6.42)	0.00 (0.01)	2.15 (2.05)
130.li	2.70 (1.94)	5.24 (3.77)	7.09 (4.64)	5.40 (3.21)	10.48 (6.25)	14.17 (7.44)
132.ijpeg	1.41 (2.22)	3.45 (3.72)	4.10 (4.94)	2.82 (4.33)	6.90 (6.95)	8.20 (8.80)
134.perl	2.06 (2.01)	4.79 (2.91)	6.29 (5.42)	4.11 (3.01)	9.59 (4.34)	12.58 (8.92)
147.vortex	1.92 (1.42)	2.80 (3.74)	11.81 (5.06)	3.84 (2.10)	5.60 (6.63)	23.63 (7.88)
101.tomcatv	3.96 (3.33)	0.63 (1.38)	5.97 (5.83)	7.93 (5.72)	1.26 (2.47)	11.92 (10.05)
102.swim	6.06 (5.09)	0.00 (0.00)	3.35 (4.45)	12.11 (8.18)	0.00 (0.00)	6.69 (6.58)
103.su2cor	7.38 (4.81)	0.44 (1.19)	2.98 (4.53)	14.76 (8.72)	0.88 (2.12)	5.98 (8.29)
107.mgrid	9.57 (2.98)	0.00 (0.02)	2.58 (1.75)	19.15 (4.41)	0.00 (0.04)	5.17 (3.00)

Table 2: Average number of data, heap, and stack accesses in the last 32 and 64 instructions. Standard deviation of the distribution is shown in the parenthesis.

data region are less bursty than accesses to the heap or stack region. Data accesses are bursty[4] in only two programs (*126.gcc* and *132.ijpeg*) while heap accesses are bursty in 8 programs and stack accesses in 6 programs when the window size is 32. When the window size is 64, however, only 3 programs (*099.go, 132.ijpeg,* and *103.su2cor*) have bursty stack accesses. This implies that when a high-performance processor captures sufficiently many instructions on-the-fly in its large instruction window, it is likely that there are some stack referencing instructions waiting to access the data cache as well as instructions that access data region.

Third, as pointed out earlier, there are few heap accesses in floating-point programs. In programs that have many heap accesses, such as *130.li, 132.ijpeg,* and *134.perl,* there are relatively fewer data accesses. It is shown that heap accesses are quite bursty even when the window size is 64. The result implies that processing heap accesses separately will not generally bring much benefit, especially for the floating-point programs.

From the above observations, it is concluded that most programs have a constant memory bandwidth demand for data and stack accesses, especially when the processor buffers many instructions to search for independent instructions to achieve higher performance.

[4]In this paper, accesses to a region are considered *bursty* if the average number of accesses in the given instruction window is smaller than the standard deviation. *132.ijpeg*'s data, heap, and stack accesses are all bursty, for instance.

3.3 A case for decoupling stack accesses

The previous subsection suggests that if an extra cache for stack references is provided, the data cache bandwidth can be saved for many programs, effectively offering more data bandwidth. The remainder of this paper focuses on servicing stack and non-stack references separately with two exclusive caches. The details of the architectural change and the effects will be discussed in Section 4.

It has been shown previously that stack accesses show a very high degree of data locality and a separate cache for stack need not be large to attain a high hit rate [4, 3]. A 4-KB stack cache was shown to achieve over 99.5% hit rate for all the SPEC95 benchmark programs studied, with an average of about 99.9% [3].

3.4 Predicting access regions

3.4.1 Dynamic access region prediction strategies

Memory instructions can be categorized into three classes: ones that always access the stack ("S" in Figure 2), ones that always access non-stack regions ("D", "H", or "D/H"), and ones that can access both the stack and non-stack regions. The goal of the discussions in this section is to develop an efficient mechanism to accurately predict which region (stack or non-stack) an instruction will access at run time, given the program counter (PC) of it. We use a hardware table called *access region prediction table* (ARPT), similar to *branch prediction table*, and a set of heuristics for higher prediction accuracy. The prediction result, available as early as at the fetch stage, is used to guide the instruction dispatcher for data decoupling as discussed in Section 4. A schematic figure to illustrate how the ARPT is operated is given in Figure 3.

Note that it is already shown that most instructions access only a single region during program execution in Figure 2. Therefore, if a single bit is allocated for each static memory instruction to store the previously accessed region, *e.g.*, using '1' to indicate stack and '0' to indicate non-stack, the access region of most instructions can be accurately predicted from next time by just looking up their history bit. This is called the *simple one-bit scheme* in this paper. Alternatively, we can use two bits per instruction to add hysteresis or inertia to prediction (*simple two-bit scheme*).

There are two issues in obtaining good prediction accuracy given the above base predictors. First, when a memory instruction is first executed, *i.e.*, the history of the instruction is not available, how do

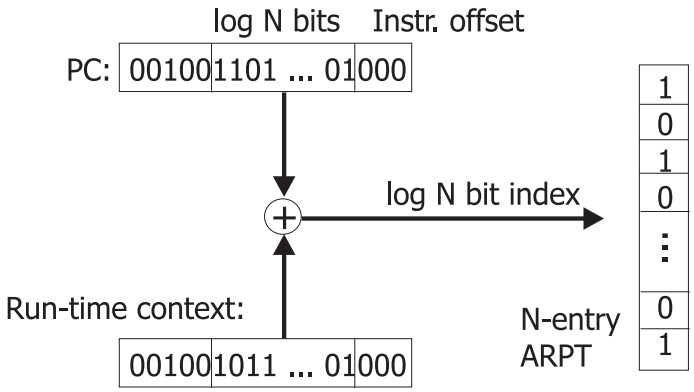

Figure 3: Operation of the ARPT with 1-bit entries. (Log N)-bits from PC (optionally XOR'ed with context bits) is used to index an N-entry ARPT.

we predict the access region of it?[5] The second issue is how we can effectively handle a memory instruction that accesses the stack ('S') and non-stack ('N') regions irregularly, *e.g.*, "SNSNSNNNSN".

To cover the above first issue, we note that the addressing mode of many memory instructions immediately reveals the regions that will be accessed. We use the following baseline heuristics in our study:[6]

(Static Prediction):

1. If the addressing mode is constant, the instruction will access a non-stack region.
2. If $sp (stack pointer) or $fp (frame pointer) is used for indexing, the instruction will access the stack region.
3. If $gp (global pointer) is used for indexing, the instruction will access a non-stack region.
4. For other cases, *i.e.*, instructions with an index register other than $sp, $fp, or $gp, *predict* that a non-stack region will be accessed.

The addressing mode information can be obtained early from the pre-decoding logic or the decoding stage at the latest. In any case, the information from the ARPT at the fetch stage will be discarded by the instruction dispatcher if the information from the (pre-)decoder is considered more accurate. Figure 4 shows that many instructions, over 50% on average, manifest their access regions in their addressing mode (lower dark bars). It is not necessary to record the history of these instructions to save

[5]Because we don't use a tag or valid bit in the ARPT, this is similar to "how do we initialize the content of the ARPT entries?"

[6]This is based on Simplescalar's *Portable ISA* (PISA) [2]. However, the discussions can be easily extended to other ISAs.

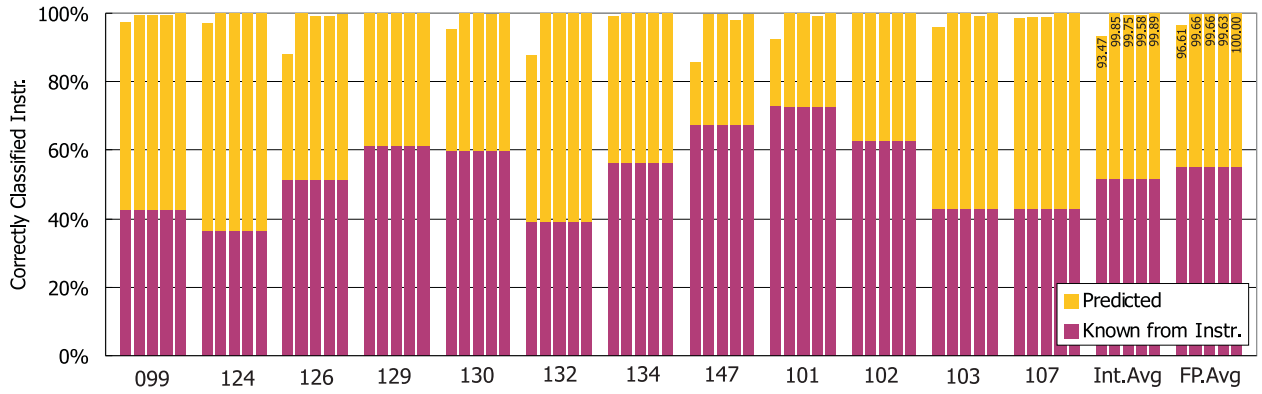

Figure 4: Percentage of memory instructions that are correctly classified into stack and non-stack regions using various schemes. Bars for each benchmark program denote the static classification, the simple 1-bit scheme, the 1-bit scheme w/ GBH context, the 1-bit scheme w/ CID, and the 1-bit scheme w/ hybrid context (8-bit GBH and 24-bit CID).

space in the ARPT. It is noted that a compiler has more knowledge of the access region of memory instructions than is disclosed in the addressing mode of them. The impact of transporting compiler information to the hardware will be studied in Section 3.5.2.

To tackle the second issue, we use an ARPT index function with additional run-time information. Figure 3 shows how the ARPT is indexed with the PC of the memory reference instruction XOR'ed with a string of bits, labeled "run-time context." We consider two types of run-time context in our work: *global branch history* (GBH) and *caller's identification* (CID). Modern branch predictors already use the GBH to index the branch prediction table [15]. The path through which the control reaches a memory instruction, like in a branch predictor, may help predict the behavior of the memory instruction. Considering which procedure called the current procedure gives a valuable insight into which region a pointered reference will access (such as *parm1 in Figure 1), because it is likely that the calling procedure passes to the called procedure the same set of arguments, or at least the arguments of the same types repeatedly. The *link register* usually keeps the next PC of the call instruction (that resides in the calling procedure) and thus can be used as a unique CID. It is also possible to combine these two types of information by concatenating a certain number of bits from the GBH and also from the CID to form a value to be used as a context.

We evaluated the following five different schemes with an unlimited ARPT: the baseline static classification (STATIC), simple 1-bit scheme (1BIT), 1-bit with GBH (1BIT-GBH), 1-bit with CID (1BIT-CID), and 1-bit with both the GBH and CID (1BIT-HYBRID). When GBH and CID are used

Benchmark	STATIC	w/ Gbh	w/ CID	w/ HYBRID
099.go	7896	9733 (23%)	13465 (71%)	34421 (336%)
124.m88ksim	1227	1319 (7%)	1326 (8%)	2093 (71%)
126.gcc	10521	11484 (9%)	15949 (52%)	30517 (190%)
129.compress	556	590 (6%)	573 (3%)	765 (38%)
130.li	822	862 (5%)	973 (18%)	1467 (78%)
132.ijpeg	3137	3467 (11%)	2681 (-15%)	5002 (59%)
134.perl	2140	2255 (5%)	2874 (34%)	4229 (98%)
147.vortex	6836	7094 (4%)	5833 (-15%)	12200 (78%)
101.tomcatv	877	954 (9%)	1033 (18%)	1430 (63%)
102.swim	958	1062 (11%)	1072 (12%)	1520 (59%)
103.su2cor	1760	2220 (26%)	3214 (83%)	5887(234%)
107.mgrid	1095	1322 (21%)	1519 (39%)	2620 (139%)

Table 3: Number of entries occupied in an unlimited ARPT, when used with different heuristics. Increase in number compared to "Static Prediction" is shown in parenthesis.

together, we use lower 8 bits from the GBH and lower 24 bits from the CID concatenated together to form a 32-bit context. Figure 4 shows the results – overall, 1BIT-HYBRID performed the best in terms of prediction rate (99.89% and 100% for the integer and floating-point programs respectively).[7]

The 1BIT scheme outperformed 1BIT-GBH and 1BIT-CID. This is because adding some run-time context in the indexing function increases the number of entries in ARPT (as shown in Table 3) and may cause unnecessary cold misses for certain memory instructions that don't need any run-time context, but still can be reached via a number of different procedures or control flows. These misses offset the benefit of using the context bits and in many cases lead to lower overall prediction rate unless the obtained benefit is large enough. *147.vortex* and *103.su2cor* are the examples where 1BIT is a clear winner over 1BIT-CID. Notice, however, that indexing with either run-time context can lead to better performance than 1BIT, as is observed in *107.mgrid.*

3.5 Performance of the access region prediction table (ARPT)

3.5.1 ARPT of limited size

We study the impact of the ARPT size on the prediction rate. Figure 5 (lower light bars) shows the performance of 1BIT-HYBRID when the ARPT size is varied from 64K to 8K entries. In general, the

[7]We omit presenting the performance of 2-bit schemes because their performance is consistently lower than that of 1-bit schemes.

Figure 5: Prediction rate achieved by 1BIT-HYBRID when the ARPT size is varied. Results for the ARPT of unlimited size, 64K, 32K, 16K, and 8K entries are shown from left. The dark, upper portion of each bar denotes performance improvement when compiler information is available.

ARPT loses the prediction accuracy when its size becomes smaller. Certain programs, such as *099.go*, *126.gcc*, *147.vortex*, and *101.tomcatv*, are more sensitive to the ARPT size than others. Table 3 indicates that these programs require significantly more entries than other programs when a type of run-time context is used (except *101.tomcatv*), putting a high pressure on the ARPT size. In the case of *101.tomcatv*, it seems that there are negative interferences in the ARPT. This is indirectly supported by the fact that the availability of compiler information removes any deficiencies caused by the limited ARPT size. It is interesting to observe that a limited ARPT performs better than an unlimited ARPT in the case of *126.gcc*. This means that there are positive interferences between different memory instructions that map to a single ARPT entry.

To summarize, a 32K-entry ARPT achieved over 99.9% prediction accuracy for both the integer and floating-point programs studied. The necessary hardware resources for implementing a 32K-entry ARPT is modest – only 4 KB of space.

3.5.2 Effects of compiler hints

A compiler often has knowledge of which region a memory instruction will access at run time. For example, b[i] and c[i] in Figure 1 are easily identifiable as heap and data references. Most stack accesses are accurately identifiable when the compiler comes across variable declarations, handles a procedure call, or performs register allocation and reloading. For non-stack accesses, the type and storage information of a variable symbol provides necessary information. A simple compiler

```
mem_type classify_mem (mem_instr* instr)
{
  if (is_local_var (instr)) return MT_STACK;
  if (is_static_var (instr)) return MT_NONSTACK;

  // Pointer deref.; assume ptr is the pointer for instr
  int flag = -1;
  for all def in UD-chain for ptr {
    if (is_function_param (def)) return MT_UNKNOWN;
    if (point_to_unknown (def)) return MT_UNKNOWN;
    if (point_to_stack (def))
      if (flag == 0) return MT_UNKNOWN;
      else flag = 1;
    if (point_to_nonstack (def))
      if (flag == 1) return MT_UNKNOWN;
      else flag = 0;
  }
  if (flag) return MT_STACK;
  return MT_NONSTACK;
}
```

Figure 6: A simple compiler algorithm to classify the access region of a memory instruction.

algorithm to determine the access region of a memory instruction is shown in Figure 6.

This section studies the effects of augmenting each static memory instruction with a tag that indicates if it is a stack access, a non-stack access, or that the compiler can not distinguish (such as *parm1 in Figure 1). For the experiments, we used profile information gathered from program runs instead of implementing the compiler algorithm and performing static analysis. The accessed region(s) of each instruction is saved in the profile data, and we assumed that an instruction can be classified by a compiler if it is shown to access only a single region during program execution. Therefore, the quality of the compiler information used in this section should be interpreted as one that is from very accurate compiler analysis (*i.e.*, upper bound). Although a real compiler will produce more unknown cases, the quality of the information will be close to the profile information in general, because memory references in many real programs are easy to analyze based on our past experiences.

The compiler information helps improve the performance of the ARPT in two ways. First, the resultant prediction rate becomes higher because many instructions have been taken care of by the compiler and bypass the prediction mechanism. Second, the space of the ARPT is saved since the correctly tagged instructions do not need to occupy the entries in the ARPT. This will decrease the effects of negative interferences and allow us to use a smaller ARPT to achieve a given prediction rate. In other words, compiler hints will reduce the pressure on the ARPT size, especially when a lot of entries are needed (*i.e.*, when a type of context bits is used).

Figure 5 (upper dark portions) shows that in the presence of compiler information (i) the prediction accuracy is higher, and (ii) the performance of the ARPT becomes less sensitive to its size. *101.tomcatv* benefits greatly from the compiler information when the ARPT has less than 32K entries. It is shown also, however, that the performance without compiler information is already high when the ARPT has 32K entries or more; Therefore, we conclude that although compiler information is useful, the hardware mechanism proposed in this paper is accurate enough for effective data decoupling as will be shown in the next section. This allows us to run all the existing binaries on a data-decoupled processor without losing noticeable performance.

4 Data-Decoupling for a Wide-Issue Superscalar Processor

4.1 Motivation

As the processors aggressively seek to widen the issue bandwidth, the complexity of several key hardware components of such processors become severe bottlenecks that hinder further scaling of the processor and shrinking the clock cycle time [18]. Especially the window logic to wake up and issue the instructions to the functional units has been pointed out to be one of the most critical hardware component that needs immediate work-arounds.

In our previous work [3], we showed that the instruction window for memory can be partitioned into two separate windows, each of which feeds a separate data cache. The approach, called data decoupling, can reduce the hardware complexity of the network and window logic required for high bandwidth memory system. Because the delay of the window logic is proportional to the square of the instruction window size and the square of the issue bandwidth [18], this partitioning can become a crucial advantage. Other architectural issues and implications are found in [3].

4.2 Pipeline design

We briefly describe the pipeline stages of a data-decoupled processor equipped with an ARPT.

- **Fetch stage.** The ARPT is accessed with the current PC XOR'ed with the run-time context bits. The single bit value returned from the ARPT is passed to the Dispatch stage.

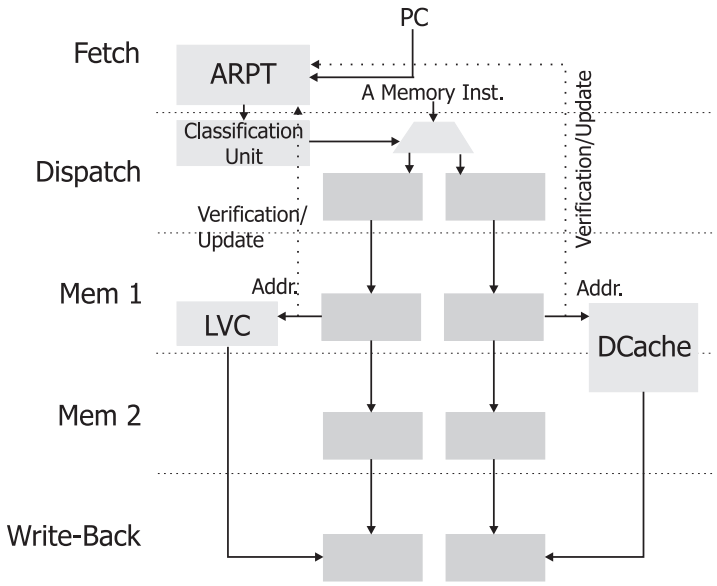

Figure 7: A processor pipeline augmented with the ARPT and a separate memory pipeline for stack and non-stack references each.

- **Dispatch (Decode) stage.** If the decoded instruction is a memory instruction, check if the addressing mode shows the access region of the instruction. If so, do not consider the ARPT bit from the Fetch stage; However, if the addressing mode does not give any hint, use the ARPT bit for deciding which of the two queues, LSQ or LVAQ, the instruction is steered into.

- **Memory access stage.** The address of a ready memory instruction in the LSQ or LVAQ is calculated in the first memory stage. Once it is known that the instruction is properly classified and placed in the right queue, the instruction will access the data cache or the local variable cache in the second memory stage. This access region checking is done when the address is translated in the TLB (Translation Look-aside Buffer). Each TLB entry is extended with a single bit indicating whether the translated page belongs to the stack or not. Storing such information can be done accurately and efficiently when a page is allocated by the run-time system. When the predicted region is determined wrong at this stage, the recovery action – be it squashing the instructions or selectively re-issuing the affected instructions – is initiated. At the same time, the ARPT is updated with correct region of the memory instruction.

4.3 Methodology and architecture model

We use a detailed timing simulator that models a superscalar processor for the results reported in this section. The simulator is based on the Simplescalar tool set [2].

The machine model used in the timing simulator supports out-of-order issue and execution, based on the *Register Update Unit* (RUU) [24]. The RUU scheme uses a reorder buffer (ROB) to automatically perform register renaming and hold the results of pending instructions. In each cycle, the ROB retires completed instructions in program order to the architected register file. The processor pipeline consists of six stages: fetch, dispatch (decode and register renaming), issue, execution, write-back, and commit. Depending on the instruction type, more than a cycle can be taken in the execution stage.

The processor's memory system employs a load/store queue (LSQ). Store values are placed in the queue if the store is speculative. Loads are dispatched to the memory system when the addresses of all previous stores are known. Loads may be satisfied either by the memory system or by an earlier store value residing in the queue. In the latter case, the store-to-load forwarding delay is one cycle.

For the experiments in this section, a model that can issue up to 16 instructions per cycle is used. The model represents a future wide-issue processor with aggressive issue bandwidth from a large instruction window. The ROB has 256 entries and the LSQ has 128 entries, which are derived from the MIPS R10000 implementation [30]. The ROB and the LSQ effectively form the instruction window of the processor. The primary on-chip data cache that is 64 KB and 2-way set-associative has a 2-cycle hit time (unless otherwise stated), as in some of the recent machines [30, 10]. The 512 KB L2 cache, either on-chip or off-chip, has a 12-cycle hit latency. Both the caches are lock-up free.

We implemented a stride-based value predictor for the register values only, as the performance improvement achieved by value prediction techniques is mainly from the register data flow [13]. The value prediction table has 16K entries in our processor model.

We use an aggressive front-end with a perfect instruction cache and a perfect branch prediction, to assert the maximum pressure on the data memory bandwidth. This setting is necessary to accurately study the impact of the proposed techniques by eliminating other factors that affect the measured performance. Important parameters of the base machine model are summarized in Table 4.

For data decoupling, the direct-mapped 1-cycle LVC is sized to 4 KB. The ARPT has 32K 1-bit entries with no tags or valid bits. The LSQ/LVAQ size was set to 96/96 entries each. On ARPT mis-

BASE MACHINE MODEL	
Issue width	16
Number of regs.	32 GPRs/32 FPRs
ROB/LSQ size	256/128
Functional units	16 integer + 16 FP ALUs, 4 integer + 4 FP MULT/DIV units.
Value predictor	Stride-based predictor for register values. 16K-entry table.
L1 data cache	2-way set-assoc. 64 KB. 2-cycle hit time. Varying number of ports.
L2 data cache	4-way set-assoc. 512 KB. 12-cycle access time.
Memory	50-cycle access time. Fully interleaved.
Local Var. Cache	Direct-mapped. 4 KB. 1-cycle access time.
ARPT	32K 1-bit entries.
Instruction cache	Perfect I-cache with a 1-cycle latency.
Branch predictor	Perfect.
Inst. latencies	Same as those of MIPS R10000 [30].

Table 4: The base machine model. Decode and commit widths are same as the issue width.

predictions, as on a branch misprediction, the instructions from the mispredicted memory instruction in the program order should be squashed and re-issued. Alternatively, only the affected instructions can be selectively re-issued with more sophisticated hardware support [13]. This paper assumes that only the dependent instructions begin to re-issue 1 cycle after the misprediction is detected.

4.4 Performance

Figure 8 shows the performance of various configurations relative to that of the (2+0) configuration.[8] The results show that the baseline configuration with a 2-ported data cache does not provide enough bandwidth: The (16+0) configuration (*i.e.*, unlimited bandwidth) improves the performance by 33% (integer) and 25% (floating-point) on average. In other words, there is room for performance improvement when more data bandwidth is provided beyond 2 ports. For *147.vortex* the improvement was as high as 80%.

Two (3+0) configurations with a 2- and 3-cycle latency achieved 21% and 18% improvement for the integer programs respectively and 14% for the floating-point programs. This shows that when the processor performance is limited by the data bandwidth, the impact of access latency becomes smaller, especially in a dynamically scheduled processor like our baseline model. Only *099.go* shows

[8] An (N+M) configuration has N-port data cache and M-port LVC. When M = 0, the configuration is a conventional memory design with an N-port data cache.

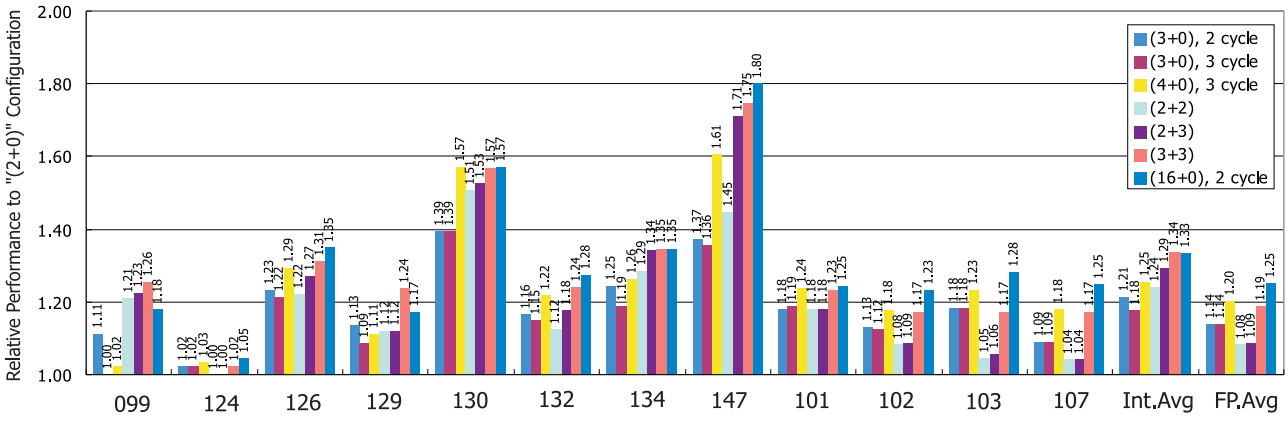

Figure 8: Performance of various configurations, compared with that of the baseline (2+0) model. The (16+0) configuration on the rightmost side is shown as an upper bound (unlimited bandwidth).

a sharp performance drop as a 3-cycle latency is used for cache access. Floating-point programs were hardly affected with the latency change, implying that the latency was effectively hidden by out-of-order execution. Having a 4-ported data cache, represented by the (4+0) configuration, improves the performance by 25% (integer) and 20% (floating-point). Considering that adding more ports beyond four gives diminishing returns, this configuration might become the choice for a conventional design. However, the hardware complexity caused by the large 128-entry LSQ and the cache can become excessive; We have accordingly set the cache access time to be 3 cycles for the configuration, not to increase the clock cycle time.

Comparing the (2+2) and (4+0) configurations, they achieved similar performance for integer programs, but the (4+0) configuration performed better for the floating-point programs. Under the (2+2) configuration, floating-point programs showed limited performance mainly because they have more demand for data region than stack. Therefore, attaching one more port to the LVC, the (2+3) configuration, doesn't help improve the performance of these programs at all. On the other hand, certain integer programs, like *126.gcc* and *147.vortex*, obtains more speedups.

The (3+3) configuration performs well for both the integer and floating-point programs. For the integer programs, this configuration was as good as the (16+0) configuration on average, shown as the limit case for our study. For the floating-point programs, the (3+3) configuration was close to the (4+0) configuration. In summary, when it is not feasible to build a single many-ported data cache attached to a large instruction window, a data-decoupled configuration, such as (3+3), can become a viable alternative that achieves a similar performance level.

5 Concluding Remarks

This paper studied an important behavior of memory access instructions, called the access region locality and how it is used to predict the region of a memory instruction accesses. Utilizing the access region locality, we showed the effectiveness of the data decoupling as a way of increasing on-chip data memory bandwidth. Following contributions are made in the paper:

- The notion of access region locality is introduced. Also given is a set of detailed profile data, showing that most memory reference instructions access only a single region at run time.
- Techniques to predict the access region of a memory instruction before the actual effective address is calculated are developed and evaluated. Results show that the proposed prediction mechanism with reasonable hardware resources can precisely determine the accessed region of an instruction (to be either stack or non-stack) with an accuracy of over 99.9% on average.
- Using a detailed cycle-by-cycle simulator, a performance study of a superscalar processor with the proposed prediction mechanism under the data decoupling model is given. Results show that the data-decoupled architecture with the proposed hardware prediction mechanism provides an adequate data bandwidth to a wide-issue processor, confirming the results of our previous work [3].

The importance of the work is that the hardware-based memory access region predictor proposed allows us to run existing binaries on a data-decoupled processor. As a future work, we will study in detail the hardware complexity of the proposed prediction mechanism and the data-decoupled memory pipeline in terms of area and cycle time. It will be also interesting to study the memory access behaviors of the programs written in C++ or Java, used extensively in the newest applications in many areas of computing.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[2] D. Burger and T. M. Austin. "The SimpleScalar Tool Set, Version 2.0," *Computer Sciences Department Technical Report*, No. 1342, Univ. of Wisconsin, June 1997.

[3] S. Cho, P.-C. Yew, and G. Lee. "Decoupling Local Variable Accesses in a Wide-Issue Superscalar Processor," *Technical Report #98-20,* Dept. of Computer Sci. and Eng., Univ. of Minnesota, May 1998.

[4] D. Ditzel and R. McLellan. "Register Allocation for Free: The C Machine Stack Cache," *Proc. of the Symp. on Architectural Support for Programming Languages and Operating Systems,* pp. 48 – 56, March 1982.

[5] EGCS Project. `http://egcs.cygnus.com`.

[6] J. Emer and D. Clark. "A Characterization of Processor Performance in the VAX-11/780," *Proc. of the 11th Int'l Symp. on Computer Architecture,* June 1984.

[7] M. J. Flynn. *Computer Architecture – Pipelined and Parallel Processor Design,* Jones and Bartlett Publishers, 1995.

[8] M. J. Flynn and L. W. Hoevel. "Execution Architecture: The DELtran Experiment," *IEEE Trans. on Computers,* C-32(2): 156 – 175, Feb. 1983.

[9] L. Gwennap. "Intel's P6 Users Decoupled Superscalar Design," *Microprocessor Report,* Vol. 9, No. 2, Feb. 1995.

[10] L. Gwennap. "Digital 21264 Sets New Standard," *Microprocessor Report,* Volume 10, Issue 14, Oct. 1996.

[11] J. Heinrich. *MIPS R10000 Microprocessor User's Manual, V1.1,* MIPS Technologies, Inc., 1995.

[12] M. Johnson. *Superscalar Microprocessor Design,* Prentice Hall, 1991.

[13] M. H. Lipasti and J. P. Shen. "Superspeculative Microarchitecture for Beyond AD 2000," *IEEE Computer,* pp. 59 – 66, Sept. 1997.

[14] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. "Value Locality and Load Value Prediction," *Proc. of the 7th Int'l Symp. on Architectural Support for Programming Languages and Operating Systems,* pp. 138 – 147, Oct. 1996.

[15] S. McFarling. "Combining Branch Predictors," WRL Technical Note TN-36, Digital Equipment Corp., June 1993.

[16] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. "Dynamic Speculation and Synchronization of Data Dependences," *Proc. of the 24th Int'l Symp. on Computer Architecture,* pp. 181 – 193, June 1997.

[17] A. Moshovos and G. S. Sohi. "Streamlining Inter-operation Memory Communication via Data Dependence Prediction," *Proc. of the 30th Annual Int'l Symp. on Microarchitecture,* pp. 235 – 245, Dec. 1997.

[18] S. Parlacharla, N. P. Jouppi, and J. E. Smith. "Complexity-Effective Superscalar Processors," *Proc. of the 24th Int'l Symp. on Computer Architecture,* pp. 206 – 218, June 1997.

[19] Y. N. Patt, S. J. Patel, D. H. Friendly, and J. Stark. "One Billion Transistors, One Uniprocessor, One Chip," *IEEE Computer,* pp. 51 – 57, Sept. 1997.

[20] D. A. Patterson and C. H. Sequin. "A VLSI RISC," *IEEE Computer*, pp. 8 – 21, Sept. 1982.

[21] J. A. Rivers, G. S. Tyson, E. S. Davidson, and T. M. Austin. "On High-Bandwidth Data Cache Design for Multi-Issue Processors," *Proc. of the 30th Annual Int'l Symp. on Microarchitecture*, pp. 46 – 56, Dec. 1997.

[22] E. Rotenberg, Q. Jacobson, and J. E. Smith. "Trace Processors," *Proc. of the 30th Annual Int'l Symp. on Microarchitecture*, pp. 138 – 148, Dec. 1997.

[23] A. J. Smith. "Cache Memories," *Computing Surveys* 14:3, pp. 473 – 530, Sept. 1982.

[24] G. S. Sohi. "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers," *IEEE Trans. on Computers*, 39(3):349 – 359, March 1990.

[25] G. S. Sohi and M. Franklin. "High-Bandwidth Data Memory Systems for Superscalar Processors," *Proc. of the Fourth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 53 – 62, April 1991.

[26] T. J. Stanley and R. G. Wedig. "A Performance Analysis of Automatically Managed Top of Stack Buffers," *Proc. of the 14th Int'l Symp. on Computer Architecture*, pp. 272 – 281, June 1987.

[27] The Standard Performance Evaluation Corporation, `http://www.specbench.org`.

[28] G. Tyson and T. M. Austin. "Improving the Accuracy and Performance of Memory Communication Through Renaming," *Proc. of the 30th Annual Int'l Symp. on Microarchitecture*, pp. 218 – 227, Dec. 1997.

[29] K. M. Wilson, K. Olukotun, and M. Rosenblum. "Increasing Cache Port Efficiency for Dynamic Superscalar Microprocessors," *Proc. of the 23th Int'l Symp. on Computer Architecture*, pp. 147 – 157, May 1996.

[30] K. C. Yeager. "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro*, Volume 16, Number 2, pp. 28 – 40, April 1996.

[31] T.-Y. Yeh, D. T. Marr, and Y. N. Patt. "Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache," *Proc. of the 7th Int'l Conf. on Supercomputing*, pp. 67 – 76, July 1993.